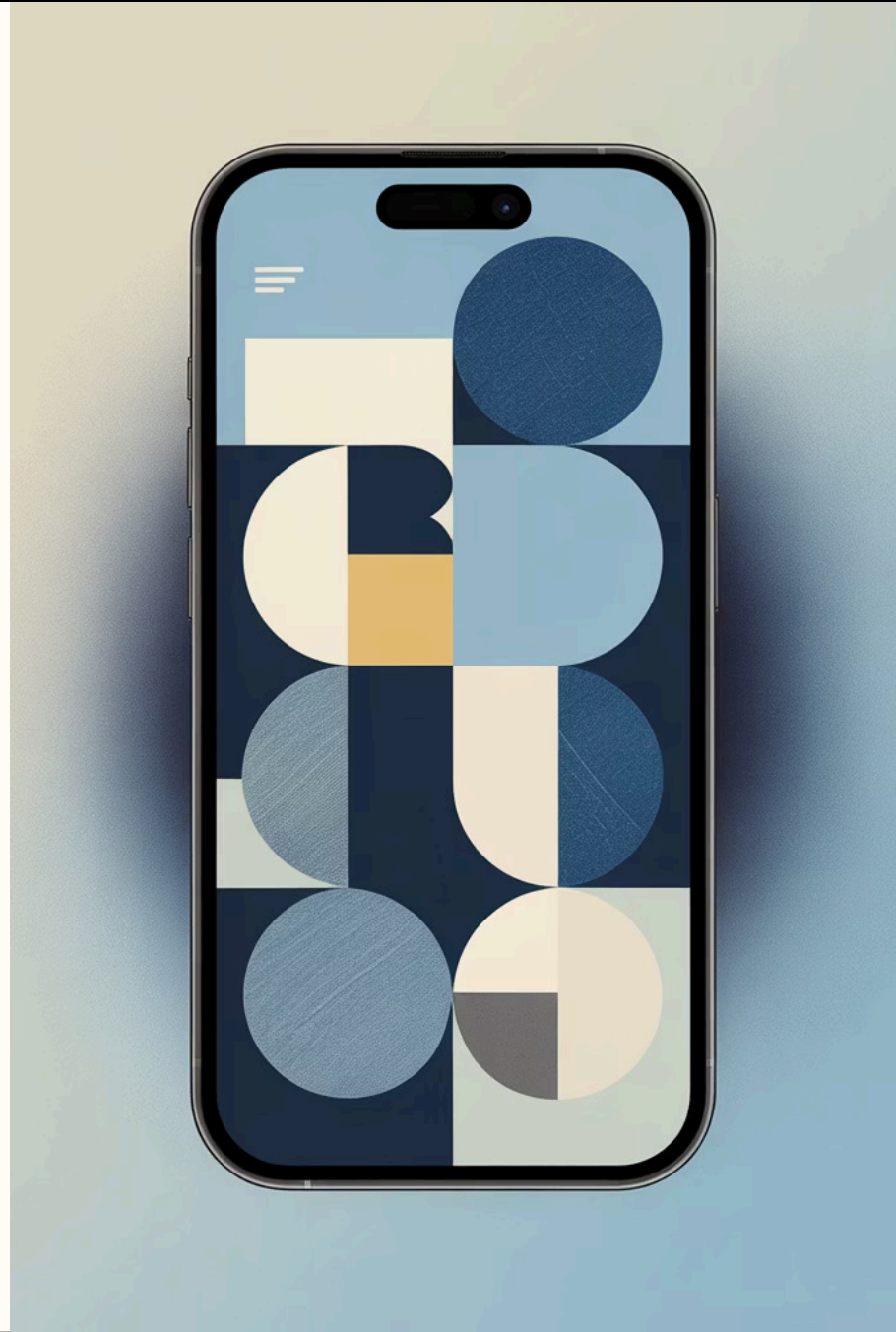
Chapitre 3 : Interface Utilisateur sous Android



Objectifs du chapitre

1. Comprendre la structure et les composants d'une interface Android
2. Manipuler les vues et layouts
3. Utiliser les listes et adaptateurs
4. Créer des interfaces adaptatives et réactives
5. Gérer les événements, menus, animations et interactions utilisateur

3.1 Insertion de Contrôles et Positionnement

Introduction aux vues (contrôles) Android

Les éléments graphiques héritent de la classe **View**. On peut regrouper des éléments graphiques dans une **ViewGroup**. Des **ViewGroup** particuliers sont prédéfinis: ce sont des gabarits (*layout*) qui proposent une prédispositions des objets graphiques:

Les déclarations se font principalement en XML, ce qui évite de passer par les instanciations Java.

Un contrôle = une vue (classe dérivée de View)

Chaque élément visible et interactif sur l'écran d'une application Android est une instance de la classe `View` ou d'une de ses sous-classes. Que ce soit un bouton, un champ de texte ou une image, tout est une "Vue".

Les contrôles sont déclarés dans le fichier XML du layout

Pour une séparation claire des préoccupations, l'interface utilisateur est généralement définie de manière déclarative dans des fichiers XML de layout, où chaque contrôle est spécifié avec ses propriétés.

Chaque contrôle possède des attributs

Ces attributs, spécifiés dans le XML, définissent l'apparence (taille, couleur, texte, arrière-plan) et le comportement initial du contrôle.

Un ID unique (@+id/...) pour être manipulé en Java

Pour interagir avec un contrôle depuis le code Java ou Kotlin (par exemple, pour modifier son texte ou écouter un clic), il doit avoir un identifiant unique.

Exemples de contrôles basiques

Pour mieux comprendre, explorons quelques-uns des contrôles les plus fondamentaux que vous utiliserez fréquemment. Ces éléments sont déclarés dans le XML et manipulables via leur ID unique.

TextView

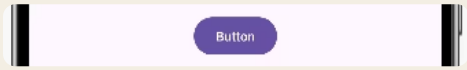
Le contrôle le plus simple pour afficher du texte statique. Il est non éditable par l'utilisateur.

- android:text : Le contenu textuel.
- android:textSize : La taille du texte.

EditText

Permet aux utilisateurs de saisir du texte, essentiel pour les formulaires et les champs de recherche.

- android:hint : Texte d'indication affiché si le champ est vide.
- android:inputType : Spécifie le type de données attendu (ex: texte, nombre).



Button

Contrôle interactif qui déclenche une action lorsqu'il est pressé.

- android:text : Le texte affiché sur le bouton.
- android:onClick : Permet de lier l'action à une méthode dans le code.

Contrôles de sélection

Les contrôles de sélection sont fondamentaux pour permettre aux utilisateurs de faire des choix dans votre application. Android propose des vues dédiées pour gérer les sélections multiples ou uniques.

CheckBox : Choix multiples

Permet à l'utilisateur de sélectionner ou de désélectionner une ou plusieurs options indépendamment (ex: Inscription aux newsletters).

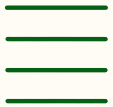
- Utilisé pour **sélectionner plusieurs** options.
- Indépendant des autres CheckBoxes.

RadioGroup + RadioButton : Choix unique

Le `RadioGroup` regroupe des `RadioButton` pour assurer un choix exclusif parmi un ensemble d'options (ex: Sexe, Méthode de paiement).

- Utilisé pour **un seul** choix parmi plusieurs.
- Les `RadioButton` doivent être contenus dans un `RadioGroup`.

Positionnement et hiérarchie des vues



LinearLayout

Organisation simple des vues en ligne (horizontale ou verticale). Idéal pour les listes et les arrangements linéaires simples.

ConstraintLayout

Positionnement flexible basé sur des contraintes relatives. Recommandé pour toutes les interfaces complexes et performantes.

FrameLayout

Utilisé pour afficher un seul élément ou superposer plusieurs vues. Idéal pour les fragments ou les superpositions d'éléments.

TableLayout

Organisation des vues en lignes et colonnes, similaire à une table HTML. Convient pour l'affichage de données tabulaires.

Attributs des Layouts

Les attributs des gabarits (Layouts) permettent d'adapter la disposition et le comportement de leurs éléments fils. Voici les plus importants :

android:layout_width et android:layout_height

Définissent la taille de l'élément dans la direction correspondante :

- "match_parent" : L'élément remplit tout l'espace disponible de son parent.
- "wrap_content" : L'élément prend la taille minimale nécessaire pour contenir son contenu.

android:orientation

Spécifie la direction d'empilement des vues enfants pour les layouts linéaires (LinearLayout).

- "vertical" : Les éléments sont empilés de haut en bas.
- "horizontal" : Les éléments sont placés côte à côte de gauche à droite.

android:gravity

Contrôle l'alignement du contenu ou des vues enfants au sein du Layout.

- "center" : Centre le contenu horizontalement et verticalement.
- "start" / "end" : Aligne le contenu au début / à la fin de la disposition.
- "top" / "bottom" : Aligne le

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:gravity="center"
  android:id="@+id/accueilid" >
    <!-- Vos contrôles ici -->
</LinearLayout>
```

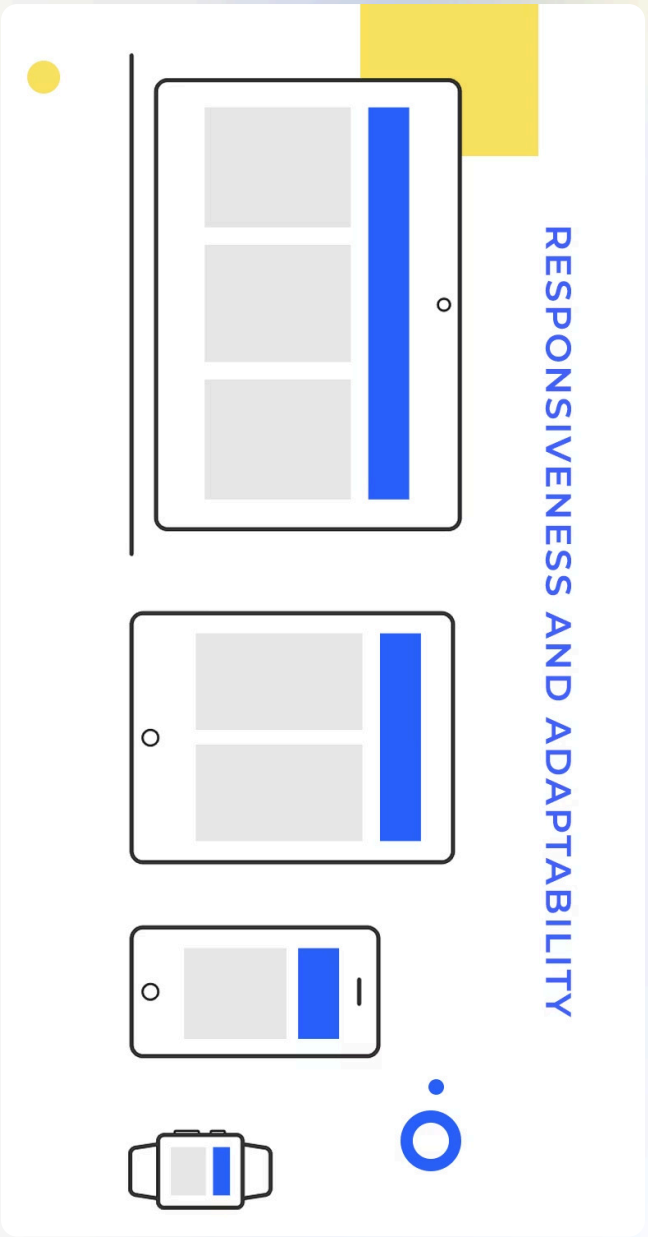
Identificateurs (ID) Android

Pour référencer et positionner un élément par rapport à un autre, les **identificateurs (ID)** sont essentiels. Ils permettent une manipulation précise des vues dans votre code.

- `android:id="@+id/idElement"` : Utilisation lors de la déclaration d'un nouvel élément dans le XML.
- `@id/idElement` : Référence à un élément existant pour le positionnement ou la manipulation.

3.2 Élasticité et Adaptabilité des UI

L'importance de créer des interfaces adaptatives qui s'ajustent aux différentes tailles d'écran et densités est primordiale dans le développement d'applications Android. Une interface utilisateur élastique et adaptable garantit une expérience utilisateur cohérente et agréable, quel que soit l'appareil utilisé, qu'il s'agisse d'un petit smartphone, d'une tablette de grande taille ou même d'un appareil pliable.



Exemple bouton responsive

```
<Button
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Envoyer" />
```

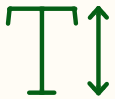
💡 Ce bouton s'adapte automatiquement à toutes les largeurs d'écran en occupant l'espace disponible (`match_parent` pour la largeur) et en ajustant sa hauteur au contenu (`wrap_content`).

Les unités de mesure

Pour garantir une interface utilisateur cohérente et adaptable sur la multitude d'appareils Android, il est crucial de comprendre et d'utiliser les bonnes unités de mesure.

dp (density-independent pixel)

Le `dp` est une unité virtuelle qui s'adapte automatiquement à la densité de l'écran. Il est l'unité préférée pour spécifier les **tailles des éléments**, les **marges** et le **padding** afin d'assurer une apparence uniforme sur différents appareils, quelle que soit leur résolution.



sp (scale-independent pixel)

Similaire au `dp`, le `sp` s'ajuste non seulement à la densité de l'écran mais aussi aux **préférences de taille de texte** de l'utilisateur (paramètres d'accessibilité). Il est donc **essentiel pour les tailles de texte** afin de garantir une meilleure lisibilité et accessibilité.

px (pixel)

Le `px` représente un **pixel réel** de l'écran. Sa taille physique varie énormément selon la densité de l'appareil, rendant les mises en page incohérentes. Il est **fortement déconseillé** pour la conception d'interfaces utilisateur adaptatives, car il ne permet pas de garantir une expérience uniforme.

Bonnes pratiques UI adaptatives

Toujours utiliser dp et sp

Utilisez les pixels indépendants de la densité (dp) pour les dimensions et les marges, et les pixels indépendants de l'échelle (sp) pour la taille du texte afin de garantir une adaptation correcte à toutes les résolutions et préférences utilisateur.

Éviter les tailles fixes

N'utilisez jamais de tailles fixes en pixels (px) car elles ne s'adaptent pas aux différentes densités d'écran, ce qui peut entraîner des interfaces cassées ou illisibles.

Utiliser ConstraintLayout pour s'adapter aux écrans

Le ConstraintLayout est le gabarit de mise en page le plus flexible pour créer des UI complexes et responsives, car il permet de définir des relations de positionnement et de taille entre les éléments.

Créer des ressources alternatives /layout-sw600dp/

Définissez des layouts et des ressources spécifiques pour différentes tailles d'écran (ex: /layout-sw600dp/ pour les tablettes) afin d'optimiser l'affichage sur chaque type d'appareil.

Tester avec l'émulateur sur différentes résolutions

Assurez-vous de tester votre application sur une variété d'émulateurs ou d'appareils réels avec différentes tailles et densités d'écran pour vérifier l'adaptabilité de votre UI.

3.3 Utilisation des Layouts

Les layouts (gabarits de mise en page) sont cruciaux pour structurer l'interface utilisateur d'une application Android. Ils agissent comme des conteneurs qui organisent et positionnent les différents éléments (vues/contrôles) de manière cohérente, garantissant ainsi une présentation visuelle harmonieuse et une expérience utilisateur optimale sur une multitude d'appareils et de tailles d'écran.

LinearLayout : Organisation linéaire

Le LinearLayout est un gabarit de mise en page qui organise les éléments (vues et contrôles) dans une seule direction : soit verticalement, soit horizontalement. Il est idéal pour des interfaces simples qui nécessitent un alignement linéaire des composants.

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    android:padding="16dp">

    <Button
        android:id="@+id/buttonHaut"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Haut"
        android:layout_gravity="center_horizontal" />
    <Button
        android:id="@+id/buttonBas"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Bas"
        android:layout_gravity="center_horizontal"
        android:layout_marginTop="8dp" />
</LinearLayout>
```

RelativeLayout : Positionnement relatif

RelativeLayout est un gabarit puissant qui permet de positionner chaque élément enfant par rapport aux bords de son conteneur parent ou par rapport à d'autres vues sœurs. Idéal pour des interfaces complexes nécessitant un positionnement flexible.

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="match_parent"    android:layout_height="match_parent">
   <Button
      android:id="@+id/button_top_left"
      android:layout_width="wrap_content"       android:layout_height="wrap_content"
      android:text="Top Left"       android:layout_alignParentTop="true"
      android:layout_alignParentStart="true"/>
   <Button
      android:id="@+id/button_center"       android:layout_width="wrap_content"
      android:layout_height="wrap_content"       android:text="Center"
      android:layout_centerInParent="true"/>
   <Button
      android:id="@+id/button_below_center"
      android:layout_width="wrap_content"       android:layout_height="wrap_content"
      android:text="Below Center"       android:layout_below="@+id/button_center"
      android:layout_centerHorizontal="true"/>

</RelativeLayout>
```

Attributs de positionnement RelativeLayout

Alignement avec le parent

Les attributs suivants permettent de positionner un élément par rapport aux bords de son conteneur parent :

- `android:layout_alignParentTop="true/false"`
- `android:layout_alignParentBottom="true/false"`
- `android:layout_alignParentStart="true/false"`
- `android:layout_alignParentEnd="true/false"`

Centrage dans le parent

Ces attributs sont utilisés pour centrer un élément au sein de son parent, soit horizontalement, soit verticalement, soit les deux :

- `android:layout_centerHorizontal="true"`
- `android:layout_centerVertical="true"`
- `android:layout_centerInParent="true"`

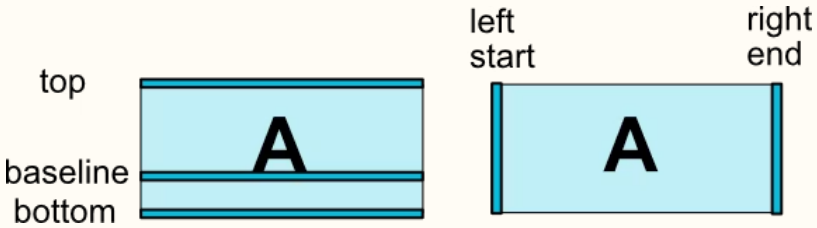
Positionnement relatif à d'autres vues

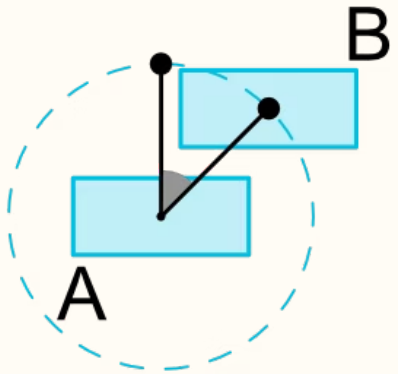
Ces attributs permettent de positionner un élément par rapport à un autre élément frère dans le même `RelativeLayout`, en utilisant son ID unique :

- `android:layout_below="@+id/idElement"`
- `android:layout_above="@+id/idElement"`
- `android:layout_toEndOf="@+id/idElement"`
- `android:layout_toStartOf="@+id/idElement"`

ConstraintLayout

Le **ConstraintLayout** est un gabarit de mise en page moderne et flexible qui permet de créer des interfaces utilisateur responsives et fluides. Il utilise un système de contraintes pour positionner et dimensionner les vues, plutôt que des positions fixes ou des relations linéaires strictes. Cela offre une grande liberté dans la conception d'interfaces complexes, capables de s'adapter à différentes tailles et orientations d'écran.





Toutes les contraintes right/left ont été remplacées par des contraintes End/Start dès API 17. End/Start facilitent l'adaptation des interfaces aux UI RTL.

Types de Contraintes dans ConstraintLayout

Contraintes Relatives

Ces contraintes positionnent une vue par rapport aux bords ou aux lignes de base d'une autre vue, offrant un contrôle fin sur l'alignement et la distribution.

- `layout_constraintStart_toStartOf`, `layout_constraintStart_toEndOf`, etc. : Positionne le bord gauche d'une vue par rapport au bord gauche ou droit d'une autre.
- `layout_constraintTop_toTopOf`, `layout_constraintTop_toBottomOf`, etc. : Positionne le bord supérieur d'une vue par rapport au bord supérieur ou inférieur d'une autre.
- `layout_constraintBaseline_toBaselineOf` : Aligne les lignes de base du texte de deux vues.

```
<Button android:id="@+id/buttonA"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="A" />
<Button android:id="@+id/buttonB"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="B"
    app:layout_constraintStart_toEndOf="@+id/buttonA" />
```

Contraintes Avancées dans ConstraintLayout

Pour des interfaces plus complexes, ConstraintLayout offre des contraintes avancées qui permettent des positionnements sophistiqués et des relations dynamiques entre les vues.

Contraintes Circulaires

- `layout_constraintCircle` : Référence l'ID de la vue centrale
- `layout_constraintCircleRadius` : Distance entre les centres
- `layout_constraintCircleAngle` : Angle de positionnement (0-360°)

Chaînes (Chains)

- Permettent de distribuer plusieurs vues de manière équilibrée
- `layout_constraintHorizontal_chainStyle` : spread, spread_inside, packed
- `layout_constraintHorizontal_weight` : Poids relatif dans la chaîne

Barrières (Barriers)

Créent une ligne de référence virtuelle basée sur plusieurs vues. Utiles pour aligner des éléments sur le bord le plus éloigné d'un groupe.

Guidelines

- Lignes de référence invisibles pour l'alignement
- Peuvent être positionnées en pourcentage ou en valeur absolue

FrameLayout

Le `FrameLayout` est le plus simple des layouts Android. Il offre une approche distincte pour l'organisation de vos vues, adapté à des scénarios d'interface utilisateur spécifiques.

FrameLayout : Superposition de vues

Le `FrameLayout` est conçu pour bloquer une zone sur l'écran afin d'afficher un seul élément. Cependant, il est très utile pour **superposer des vues les unes sur les autres**, ce qui permet de créer des effets de calque ou de placer un bouton sur une image, par exemple.

- Idéal pour les **fragments** ou les **superpositions**.
- Les vues sont empilées au **coin supérieur gauche** par défaut.

TableLayout : Structure en lignes et colonnes

Le TableLayout organise les vues en lignes et colonnes, comme une table HTML. Chaque ligne est un TableRow, et chaque cellule peut contenir une vue unique. Il est parfait pour afficher des données tabulaires ou des formulaires.

```
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="match_parent" android:layout_height="wrap_content" android:stretchColumns="1">
   <TableRow>
      <TextView android:layout_width="wrap_content" android:layout_height="wrap_content"
         android:text="Nom :"          android:padding="5dp" />
      <EditText android:layout_width="match_parent" android:layout_height="wrap_content"
         android:hint="Entrez votre nom" />
   </TableRow>
   <TableRow>
      <TextView android:layout_width="wrap_content" android:layout_height="wrap_content"
         android:text="Email :"android:padding="5dp" />
      <EditText android:layout_width="match_parent" android:layout_height="wrap_content"
         android:hint="Entrez votre email" android:inputType="textEmailAddress" />
   </TableRow>
   <TableRow>
      <TextView android:layout_width="wrap_content" android:layout_height="wrap_content"
         android:text="Téléphone :" android:padding="5dp" />
      <EditText android:layout_width="match_parent" android:inputType="phone"
         android:layout_height="wrap_content" android:hint="Entrez votre téléphone"/>
   </TableRow></TableLayout>
```

3.4 Familles de Composants

Les composants Android, éléments fondamentaux de toute application, peuvent être regroupés en différentes familles selon leur fonction et leur rôle au sein de l'interface utilisateur et de la logique applicative. Cette structuration permet une meilleure organisation et une gestion efficace du développement. On distingue principalement les familles de composants **textuels** pour l'affichage d'informations, **interactifs** pour l'interaction avec l'utilisateur, **visuels** pour la présentation graphique, et les **conteneurs** pour organiser et grouper d'autres composants.

Les principales familles de View

Textuels

TextView, EditText, AutoCompleteTextView

Interactifs

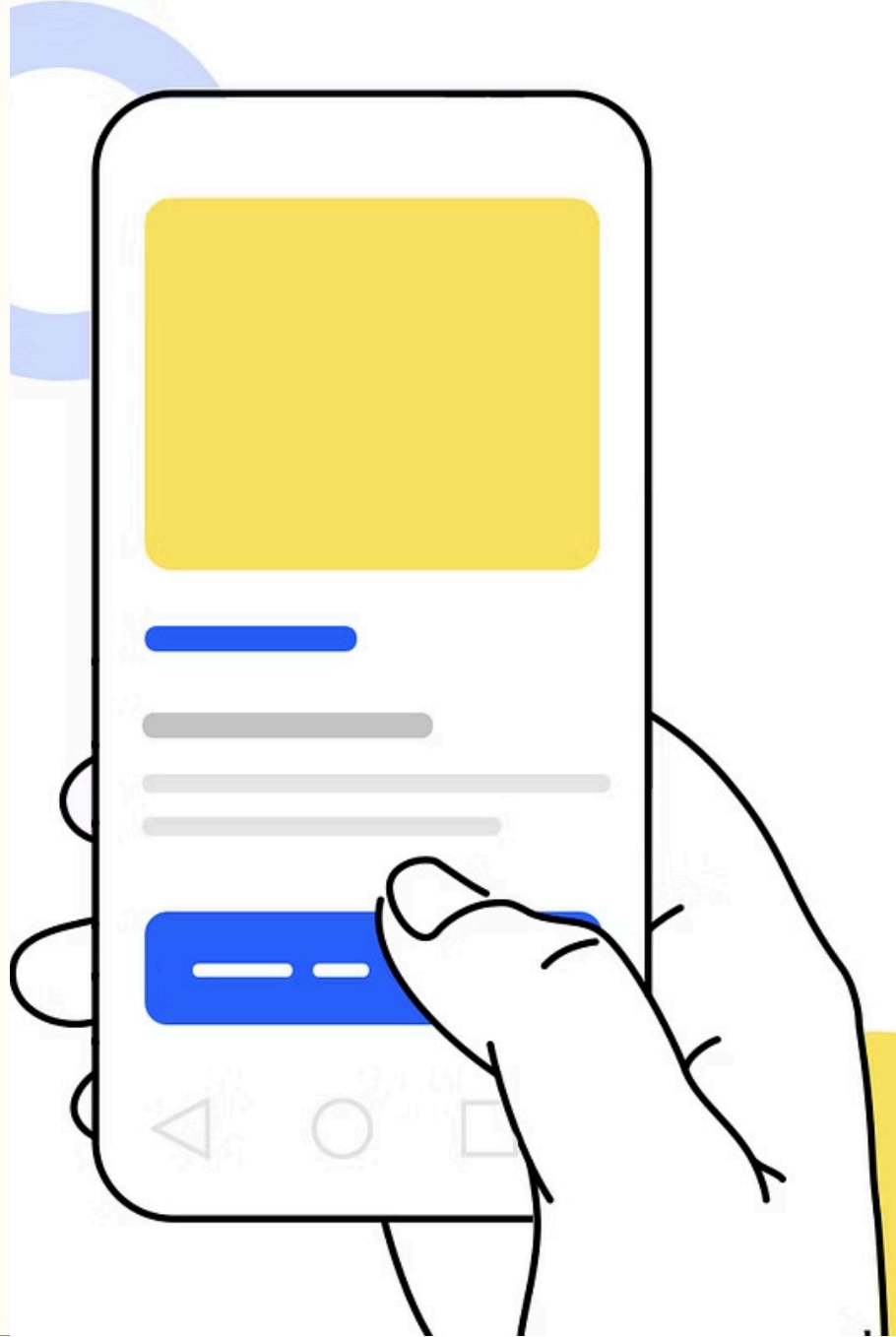
Button, Switch, CheckBox, RadioButton

Visuels

ImageView, ProgressBar

Conteneurs

LinearLayout, FrameLayout, ConstraintLayout



Composants complexes

Pour des interfaces utilisateur plus avancées et dynamiques, Android propose une gamme de composants complexes qui facilitent la gestion des listes, le défilement, la navigation et les barres d'action. Ces composants sont essentiels pour construire des applications modernes et réactives.

RecyclerView

Le RecyclerView est un composant puissant pour afficher de grandes listes de données ou des grilles de manière efficace. Il recycle les vues à mesure qu'elles sortent de l'écran, ce qui améliore considérablement les performances, surtout pour les listes très longues.

ScrollView

Le ScrollView permet de rendre l'intégralité de son contenu défilable verticalement. Il est utile lorsque le contenu d'un écran est plus grand que la zone d'affichage disponible, assurant que l'utilisateur peut accéder à toutes les informations.

ViewPager2

ViewPager2 est une version améliorée du ViewPager, facilitant la navigation par onglets avec des gestes de balayage (swipe) entre différentes vues ou fragments. Il est couramment utilisé pour les introductions d'applications ou les sections avec des onglets.

Toolbar / AppBarLayout

La Toolbar est une barre d'action flexible qui remplace l'ancienne ActionBar. Associée à l'AppBarLayout, elle permet de créer des barres d'outils riches en fonctionnalités, supportant les titres, les menus et les icônes de navigation, tout en intégrant le défilement et les réactions aux gestes de l'utilisateur.

3.4 Listes et Adaptateurs

Listes et Adaptateurs : Définition

Les Adaptateurs

Un adaptateur fait le lien entre une source de données et une vue (liste, grille, etc.), permettant d'afficher des données complexes de manière efficace dans des composants d'interface utilisateur.

- **Rôle** : Il convertit les données d'un format spécifique à un autre format compréhensible par un widget d'affichage, comme un `ListView` ou `RecyclerView`.
- **Nécessité** : Ils sont essentiels pour séparer la logique de présentation des données (la "vue") de la logique métier qui gère les données elles-mêmes, favorisant une architecture propre et maintenable.
- **Principe du pattern Adapter** : Permet à des interfaces incompatibles de collaborer. Il s'interpose entre les données brutes et le composant UI, adaptant les données pour l'affichage.
- **Types courants** : On retrouve souvent `ArrayAdapter` (pour les données simples comme des chaînes de caractères), `BaseAdapter` (pour les vues personnalisées) et `RecyclerView.Adapter` (pour des listes plus performantes et flexibles).
- **Avantages** : Permet la réutilisabilité du code (l'adaptateur peut être utilisé avec différentes vues si les données sont compatibles) et améliore la performance en gérant intelligemment le recyclage des vues, évitant ainsi de recréer des éléments graphiques inutilement.

Exemple avec ListView

```
ListView listView = findViewById(R.id.listView);
String[] villes = {"Rabat", "Agadir", "Safi"};
ArrayAdapter<String> adapter = new ArrayAdapter<>(this, android.R.layout.simple_list_item_1, villes);
listView.setAdapter(adapter);
```

L'`ArrayAdapter` convertit chaque élément du tableau `villes` en une ligne de texte simple affichée dans la `ListView`.

RecyclerView

Le `RecyclerView` est la version moderne et **plus performante** que `ListView` pour l'affichage de grandes collections de données. Il offre une meilleure gestion de la mémoire et une fluidité accrue grâce à son architecture modulaire.

Adapter

Définit la structure des éléments et assure la liaison entre les données et les vues.

ViewHolder

Réutilise les vues existantes pour un élément de liste, optimisant ainsi les performances de défilement.

LayoutManager

Définit la manière dont les éléments sont disposés (liste linéaire, grille ou cascade).

Exemple RecyclerView

Pour mettre en œuvre un `RecyclerView` dans votre activité, vous devez l'initialiser, lui assigner un `LayoutManager` et un `Adapter`.

```
// Dans MainActivity.java
RecyclerView recyclerView = findViewById(R.id.my_recycler_view);
recyclerView.setLayoutManager(new LinearLayoutManager(this));
recyclerView.setAdapter(new MonAdapter(vosDonnees));
```

Ces lignes de code clés initialisent le `RecyclerView`, définissent comment ses éléments seront disposés (ici, une liste linéaire verticale) et le connectent à un `Adapter` personnalisé qui fournit les données et gère la création des vues.

L'adaptateur est la clé pour afficher vos données dans le `RecyclerView`. Il gère la création des vues et leur association aux données.

```
public class MonAdapter extends RecyclerView.Adapter<MonAdapter.MonViewHolder> {
    private String[] donnees;
    public MonAdapter(String[] donnees) { this.donnees = donnees; }
    @Override
    public MonViewHolder onCreateViewHolder(@NonNull ViewGroup parent, int viewType) {
        View view = LayoutInflater.from(parent.getContext()).inflate(R.layout.item_layout, parent, false);
        return new MonViewHolder(view);
    }
    @Override
    public void onBindViewHolder(@NonNull MonViewHolder holder, int position) {
        holder.textView.setText(donnees[position]);
    }
    @Override
    public int getItemCount() {      return donnees.length;    }
    public static class MonViewHolder extends RecyclerView.ViewHolder {
        public TextView textView;
        public MonViewHolder(@NonNull View itemView) {
            super(itemView);
            textView = itemView.findViewById(R.id.item_text_view);
        }
    }
}
```

Layout XML pour les items

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    android:padding="16dp">

    <TextView
        android:id="@+id/item_text_view"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="18sp"
        android:textColor="@android:color/black" />

</LinearLayout>
```

Ce fichier XML définit la structure visuelle d'un seul élément (item) qui sera affiché dans une liste (comme un `RecyclerView` ou `ListView`). Ici, un simple `TextView` est utilisé pour afficher du texte pour chaque élément.

Autres types de listes et conteneurs

ExpandableListView

Permet d'afficher des données sous forme de groupes qui peuvent être étendus ou réduits pour révéler des éléments enfants, idéal pour les structures hiérarchiques.

ListActivity / ListFragment

Des classes utilitaires qui simplifient la création d'écrans entièrement dédiés à l'affichage d'une liste, avec des méthodes pré-configurées pour la gestion des adaptateurs.

RecyclerView

La solution moderne et performante pour les listes volumineuses ou complexes, offrant un meilleur contrôle sur l'affichage et le recyclage des vues pour une fluidité optimale.

StackView

Affiche les éléments sous forme de pile de cartes, permettant un défilement vertical ou horizontal avec un effet de rotation, idéal pour des présentations visuelles.

Spinner

Une liste déroulante interactive qui permet à l'utilisateur de choisir une option unique parmi un ensemble de choix, similaire à une balise `<select>` HTML.

CardView

Un conteneur léger offrant un aspect de "carte" avec des coins arrondis et une ombre portée. Il est utilisé pour présenter des informations de manière visuellement distincte.

3.4 Changements d’Activités et Vues

La navigation et le transfert de données sont fondamentaux pour créer des applications Android interactives et dynamiques. Un `Intent` est le mécanisme principal pour orchestrer ces interactions.

Navigation entre activités

La navigation entre différentes "activités" (écrans) est une fonctionnalité fondamentale dans les applications Android. Un **Intent** est l'objet qui permet cette transition.

Un **Intent** est un message abstrait qui permet de démarrer une autre activité, un service, ou de diffuser un événement. Il agit comme un messager entre les composants de l'application ou même entre différentes applications. Il peut transporter des données entre les activités.

```
Intent i = new Intent(this, SecondActivity.class);
startActivity(i);
```

Dans cet exemple, un `Intent` est créé pour lancer `SecondActivity` à partir de l'activité courante (représentée par `this`). La méthode `startActivity(i)` exécute cet Intent et démarre la nouvelle activité.

Transfert de données entre activités

Le transfert de données est crucial pour que les activités puissent interagir et partager des informations. L'objet **Intent**, en plus de lancer des activités, permet également de transporter des données d'une activité à l'autre grâce à la méthode `putExtra()`.

Pour envoyer des données, vous utilisez `putExtra()` sur votre `Intent`. Cette méthode prend une clé (une chaîne de caractères) et la valeur que vous souhaitez transférer. Les types de données supportés incluent les primitives (int, boolean, float, etc.), les chaînes de caractères (String), et les objets sérialisables ou parcelables.

```
Intent i = new Intent(this, SecondActivity.class);
i.putExtra("nom", "Ali");
startActivity(i);
```

Dans l'activité de destination (`SecondActivity` dans cet exemple), vous pouvez récupérer les données envoyées via l'objet `Intent` qui a servi à la lancer. Pour cela, vous utilisez des méthodes comme `getStringExtra()`, `getIntExtra()`, etc., en spécifiant la même clé que celle utilisée lors de l'envoi.

```
String nom = getIntent().getStringExtra("nom");
```

Dans cet exemple, la chaîne de caractères "Ali" est associée à la clé "nom" et envoyée à `SecondActivity`. Là-bas, la valeur est récupérée en utilisant `getIntent().getStringExtra("nom")`, ce qui attribue "Ali" à la variable `nom`.

3.5 Gestion des Événements

La gestion des événements est au cœur de toute application Android interactive. Elle permet à votre application de réagir aux actions de l'utilisateur, comme les clics sur un bouton, les saisies de texte ou les changements d'état des composants UI. Comprendre ce mécanisme est essentiel pour créer une expérience utilisateur fluide et dynamique.

Exemple: Événements de Clic (OnClickListener)

Le gestionnaire d'événements le plus fondamental en Android est l'`OnClickListener`, utilisé pour détecter les clics des utilisateurs sur divers composants d'interface utilisateur comme les boutons, les images ou d'autres vues interactives.

Il est essentiel pour rendre votre application réactive aux interactions tactiles, permettant d'exécuter du code spécifique lorsqu'un utilisateur appuie sur un élément.

```
// Dans votre activité ou fragment
Button monBouton = findViewById(R.id.mon_bouton);
monBouton.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        // Code à exécuter lorsque le bouton est cliqué
        Toast.makeText(MonActivite.this, "Bouton cliqué !", Toast.LENGTH_SHORT).show();
    }
});
```

Cet exemple simple montre comment associer un `OnClickListener` à un bouton. Dès que l'utilisateur appuie sur "monBouton", la méthode `onClick()` est déclenchée, affichant un message temporaire (`Toast`) à l'écran.

Exemple : Événements de Focus (OnFocusChangeListener)

Les événements de focus sont essentiels pour gérer l'interaction des utilisateurs avec des champs de saisie de texte ou d'autres éléments interactifs. L'interface `OnFocusChangeListener` permet de détecter quand un composant d'interface utilisateur gagne ou perd le focus.

Cette capacité est particulièrement utile pour la validation de formulaires en temps réel, par exemple pour vérifier le format d'une adresse e-mail dès que l'utilisateur quitte le champ de saisie, ou pour modifier l'apparence d'un champ lorsqu'il est actif.

```
EditText monChampTexte = findViewById(R.id.mon_champ_texte);
monChampTexte.setOnFocusChangeListener(new View.OnFocusChangeListener() {
    @Override
    public void onFocusChange(View v, boolean hasFocus) {
        if (hasFocus) { // Le composant a gagné le focus
            monChampTexte.setBackgroundColor(getResources().getColor(R.color.focused_background));
        } else {// Le composant a perdu le focus
            monChampTexte.setBackgroundColor(getResources().getColor(R.color.default_background));
            // Ici, vous pourriez déclencher une validation pour ce champ
            if (monChampTexte.getText().toString().isEmpty()) {
                monChampTexte.setError("Ce champ ne peut pas être vide.");
            }
        }
    }
});
```

Exemple: Événements de Saisie (TextWatcher)

Pour détecter et réagir aux changements dans un champ de saisie de texte (`EditText`), Android utilise l'interface `TextWatcher`. Cette interface fournit des méthodes qui sont appelées avant, pendant et après que le texte d'un `EditText` ait été modifié par l'utilisateur.

L'implémentation d'un `TextWatcher` est essentielle pour des fonctionnalités comme la validation de formulaire en temps réel, la recherche instantanée ou l'adaptation de l'interface utilisateur en fonction du contenu saisi.

```
// Dans votre activité
EditText monEditText = findViewById(R.id.mon_edit_text);
monEditText.addTextChangedListener(new TextWatcher() {
    @Override
    public void beforeTextChanged(CharSequence s, int start, int count, int after) { ... }

    @Override
    public void onTextChanged(CharSequence s, int start, int before, int count) {
        Log.d("TextWatcher", "Pendant modification: " + s.toString());
    }

    @Override
    public void afterTextChanged(Editable s) { ... }
})
```

3.6 Les Menus

La gestion des menus est un aspect important de l'interface utilisateur Android. Les menus permettent d'organiser les actions et options de votre application de manière claire et accessible.

Menus d'options

Les menus d'options sont définis dans des fichiers XML stockés dans le dossier `/res/menu/`. Ils permettent de regrouper les actions principales de votre application dans un menu accessible depuis la barre d'action.

Fichier XML dans `/res/menu/main_menu.xml` :

```
<menu>
    <item android:id="@+id/action_settings"
        android:title="Paramètres" />
</menu>
```

Ce fichier XML définit un menu simple avec un seul élément "Paramètres". Chaque item possède un ID unique et un titre affiché à l'utilisateur.

Lier le menu à l'activité

Pour afficher le menu dans votre activité, vous devez surcharger la méthode `onCreateOptionsMenu()` dans votre classe Activity. Cette méthode est appelée automatiquement par Android lors de la création du menu.

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.main_menu, menu);
    return true;
}
```

La méthode `getMenuInflater().inflate()` charge le fichier XML du menu et l'ajoute à l'interface utilisateur. Le retour true indique à Android que le menu doit être affiché.

Menus contextuels

Les menus contextuels apparaissent lorsque l'utilisateur effectue un clic long sur un élément de l'interface. Ils sont particulièrement utiles pour proposer des actions spécifiques à un élément donné.

Exemple d'utilisation :

- Supprimer un item d'une liste
- Modifier un élément sélectionné
- Partager du contenu

Les menus contextuels offrent une expérience utilisateur intuitive en proposant des actions pertinentes selon le contexte d'utilisation, sans encombrer l'interface principale.

Implémentation d'un Menu Contextuel

Pour mettre en place un menu contextuel, vous devez d'abord définir sa structure en XML, puis l'associer à une vue spécifique dans votre activité et enfin gérer les actions de ses éléments.

1. Définition du menu contextuel (`res/menu/context_menu.xml`**) :**

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/action_edit"
        android:title="Modifier" />
    <item android:id="@+id/action_delete"
        android:title="Supprimer" />
</menu>
```

Ce fichier XML crée un menu simple avec deux options : "Modifier" et "Supprimer".

Intégration du menu contextuel dans l'activité

2. Intégration dans l'activité (VotreActivity.java) :

```
// Dans votre méthode onCreate() ou autre méthode d'initialisation
TextView myTextView = findViewById(R.id.my_text_view);
registerForContextMenu(myTextView); // Associe le menu au TextView

// Surchargez onCreateContextMenu pour gonfler le menu
@Override
public void onCreateContextMenu(ContextMenu menu, View v, ContextMenu.ContextMenuInfo menuInfo) {
    super.onCreateContextMenu(menu, v, menuInfo);
    if (v.getId() == R.id.my_text_view) {
        getMenuInflater().inflate(R.menu.context_menu, menu);
    }
}
```

Cette étape associe le menu contextuel à une vue spécifique (ici un TextView) et définit comment le menu est créé lorsque l'utilisateur effectue un appui long.

Gestion des actions du menu

3. Gestion des actions (onContextItemSelected) :

```
// Surchargez onContextItemSelected pour gérer les clics sur les éléments du menu
@Override
public boolean onContextItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.action_edit:
            Toast.makeText(this, "Action Modifier", Toast.LENGTH_SHORT).show();
            return true;
        case R.id.action_delete:
            Toast.makeText(this, "Action Supprimer", Toast.LENGTH_SHORT).show();
            return true;
        default:
            return super.onContextItemSelected(item);
    }
}
```

Dans cet exemple, le `TextView` avec l'ID `my_text_view` déclenchera le menu contextuel lors d'un appui long. La méthode `onContextItemSelected` gère les actions de l'utilisateur selon l'élément sélectionné.

3.7 Dialogue avec l'Utilisateur

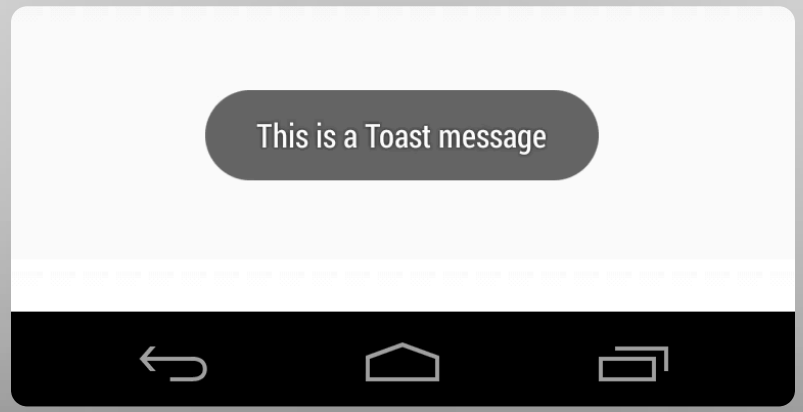
La communication avec l'utilisateur est un aspect fondamental du développement d'applications Android. Les dialogues permettent d'informer, de demander confirmation ou de recueillir des informations de manière interactive et non intrusive.

Toast

```
Toast.makeText(this, "This is a Toast Message",
Toast.LENGTH_SHORT).show();
```

✅ Affichage temporaire, utile pour feedback rapide.

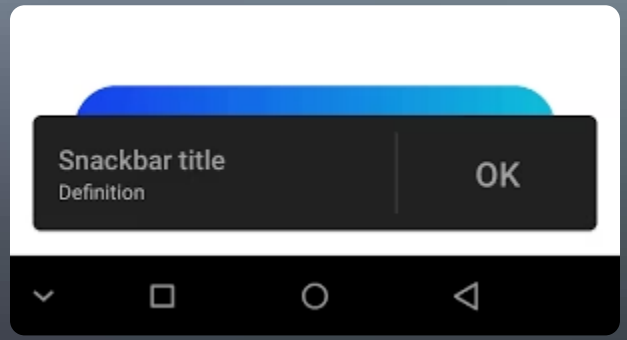
Le Toast est le moyen le plus simple d'afficher un message bref à l'utilisateur. Il apparaît temporairement à l'écran sans interrompre l'interaction avec l'application.



AlertDialog

```
new AlertDialog.Builder(this)
    .setTitle("Confirmation")
    .setMessage("Voulez-vous quitter ?")
    .setPositiveButton("Oui", (d, i) -> finish())
    .setNegativeButton("Non", null)
    .show();
```

L'AlertDialog est un composant puissant pour créer des boîtes de dialogue modales qui demandent une action de l'utilisateur. Il peut contenir un titre, un message et jusqu'à trois boutons d'action.



Snackbars (modernes)

```
Snackbar.make(findViewById(R.id.main), "Données enregistrées", Snackbar.LENGTH_SHORT).show();
```

💡 Recommandé par Material Design pour les messages contextuels.

Les Snackbars sont la solution moderne recommandée par Google pour afficher des messages temporaires. Contrairement aux Toast, ils peuvent inclure des actions et s'intègrent parfaitement dans le design Material.